PostgreSQL Server Development

Stephen Frost
stephen@crunchydata.com

Crunchy Data

October 27, 2015

Trainer - Stephen Frost

- Chief Technology Officer @ Crunchy Data
- Committer
- Major Contributor
- Implemented the roles system in 8.3
- Column-level privileges in 8.4
- Contributions to PL/pgSQL, PostGIS

Crunchy Data

What is Crunchy Data?

- PostgreSQL Support
- Training
- Consulting
- Open Source Development
- Committed to Open Source

`http://crunchydata.com`

Official Release tarball

- Only the sources from the release
- Includes parser output (bison/flex)
- Fewer dependencies required for building
- Not very useful for developing though

```
wget http://postgresql.org/ ...
```

Using git

- Clone down the PostgreSQL public git repository
- Contains all of the changes to PostgreSQL
- Since original CVS import in 1996
- Postgres95 1.01 Distribution ("Virgin Sources")
- Around 40,000 commits to "master" since
- Already done on VM - /home/training/pg/src

```
cd /home/training/pg/src/master
git clone git://git.postgresql.org/git/postgresql.git
```

Building from git

- Additional requirements to build
- Included on VM
- bison/flex
- Let's build it!

```
cd /home/training/pg
./build.sh master
```

Build script

- Guts of the build script:

```
(cd "$BUILD" && \
 CFLAGS=-I/usr/include/mit-krb5 \
 LDFLAGS=-L/usr/lib/x86_64-linux-gnu/mit-krb5 \
 "$SOURCE"/configure --silent --prefix="$INSTALL" --with-openssl \
    --with-tcl --with-tclconfig=/usr/lib/tcl8.6 --with-perl \
    --enable-debug --enable-cassert --enable-tap-tests --with-gssapi && \
 make -s -j5 && \
 make -s -j5 install && \
 make -s -j5 check && \
 make -s -j5 world && \
 make -s install-world && \
 make -s check && \
 make -s check-world \
)
```

Build script

- build.sh includes my personal options
- Very similar to Debian/Ubuntu build
- Completely clean build (old builds rm -rf'd)
- Parallel (-j5 build)
- Silent configure (–silent) and build (make -s)
- Warning/errors will still be displayed
- Built with debugging and assertions
- Builds/installs/checks "world"

Building PostgreSQL

- Targets:
  - Default / "all" - Just builds base PostgreSQL
  - "check" - Runs base PostgreSQL regression test
  - "install" - Installs base PostgreSQL
  - "world" - Build PostgreSQL + extensions + documentation
  - "check-world" - Runs extension regression tests too
  - "install-world" - Installs PG, docs, extensions
  - "installcheck" - Runs regression test against *existing* PG
  - "installcheck-world" - + Extension tests against existing PG

Building PostgreSQL - Requirements

- Targets:
  - Base build requires normal build dependencies
  - Plus bison/flex
  - Documentation depends on OpenJade, docbook
  - Lots of options, may require additional dependencies
  - PL/Perl requires perl, libperl-dev, etc
  - LDAP support requires libldap2-dev
  - VM installed with all build dependencies for Debian-based

Commit Log

- Every copy of git repository contains all changes
- Does not require network to access/review log

```
cd /home/training/pg/src/master
git log

commit .....
Author: ...
Date: ...

  Commit title

  Commit description/log
```

Commit Log

- Commit logs as important, or maybe more, than comments
- Why is extremely imporant!
- Able to filter based on author/committer

```
cd /home/training/pg/src/master
git log --author sfrost

commit b7aac36245261eba9eb7d18561ce44220b361959
Author: Stephen Frost <sfrost@snowman.net>
Date:   Fri Oct 9 10:49:02 2015 -0400

    Handle append_rel_list in expand_security_qual

    During expand_security_quals, we take the security barrier quals on an
    RTE and create a subquery which evaluates the quals.  During this, we
    [...]
```

PostgreSQL Committer vs. Author

- PostgreSQL does not track "author" using git
- For PG, in git, "Author" and "Committer" always the same
- Authors and contributors mentioned in commit log, eg:

```
commit b7aac36245261eba9eb7d18561ce44220b361959
Author: Stephen Frost <sfrost@snowman.net>
Date:   Fri Oct 9 10:49:02 2015 -0400

    Handle append_rel_list in expand_security_qual

[...]

Patch by Dean Rasheed
```

PostgreSQL Backpatching

- Generally noted in commit log if patch is back-patched

```
commit be400cd25c7f407111b9617dbf6a5fae761754cb
Author: Stephen Frost <sfrost@snowman.net>
Date:   Mon Oct 5 10:14:49 2015 -0400

    Add regression tests for INSERT/UPDATE+RETURNING

    This adds regressions tests which are specific to INSERT+RETURNING and
    UPDATE+RETURNING to ensure that the SELECT policies are added as
    WithCheckOptions (and should therefore throw an error when the policy is
    violated).

    Per suggestion from Andres.

    Back-patch to 9.5 as the prior commit was.
```

Overview

- Branches maintained for major versions
- Only bug-fixes go into released versions
- Feature development happens against master
- Occationally, features back-patched to next release, pre-beta
- Branch list (-r to show remotes):

Listing Branches

```
git branch -r
  origin/HEAD -> origin/master
  origin/REL2_0B
  origin/REL6_4
  ...
gbr
  origin/HEAD -> origin/master
  origin/REL2_0B
  origin/REL6_4
  ...
gb -r
  origin/HEAD -> origin/master
  origin/REL2_0B
  origin/REL6_4
  ...
gb
  ...
  REL9_5_STABLE
  feature
  master
```

Work on a local branch

- PostgreSQL minimizes number of upstream branches
- Ongoing development works through email and patches
- Local branches are encouraged to allow frequent commits
- Changes will be "squashed"/"merged" before posting
- Single, complete, generally seen as easier to review

Feature Branch

- Feature branch created on VM already
- VM uses multi-work-dir git feature (more later)
- One directory per branch
- Simplifies working with multiple branches
- Branches checked out on VM:

```
cd /home/training/pg/src
ls -1
  feature
  make_branch.sh
  master
  REL9_1_STABLE
  REL9_2_STABLE
  REL9_3_STABLE
  REL9_4_STABLE
  REL9_5_STABLE
```

Feature Branch

- Feature branch currently identical to master
- git log -1; shows just last commit

```
cd /home/training/pg/src/master
git log -1
cd /home/training/pg/src/feature
git log -1
```

More later...

- We will cover more on branches later...

Jumping into the source tree

- Components of PostgreSQL:
  - psql - Command-line tool, client-side
  - libpq - Client-side library, used by psql (and others)
  - bin - Other binaries (Mostly server-side- initdb, etc)
  - backend - PostgreSQL Server-side code
  - contrib - Extensions to PostgreSQL

Code Style

- Try to make your code 'fit in'
- Follow the PG style guide in the FAQ
- Beware of copy/paste
- Only C-style comments
- Comments go on their own lines, generally
- In comments, talk about why, not what or how
- Comment blocks for functions, loops, etc

psql source

- psql lives in src/bin/psql
- View in our feature branch:

```
cd /home/training/pg/src/feature
cd src/bin/psql
```

Main Components of psql

- startup.c - main(), option parsing, psqlrc, etc
- mainloop.c - Reads input, sends commands to backend
- command.c - Handle backslash commands
- describe.c -

  `All describe (\d) commands`
- tab_complete.c - Tab completion, very handy

Other Components of psql

- copy.c -

  `Handle \copy requests`
- large_obj.c - Handle large objects (PG LO, not bytea)
- mbprint.c - Multibyte character handling
- help.c - Various help/usage routines
- print.c - Output/query result handling
- input.c - User-entered info, readline interface, history
- prompt.c - Constructs user-defined psql prompt
- common.c - error/cancel handling, -o support, backend queries

Other Components of psql

- copy.c -

  `Handle \copy requests`
- large_obj.c - Handle large objects (PG LO, not bytea)
- mbprint.c - Multibyte character handling
- help.c - Various help/usage routines
- print.c - Output/query result handling
- input.c - User-entered info, readline interface, history
- prompt.c - Constructs user-defined psql prompt
- common.c - error/cancel handling, -o support, backend queries

Add a new backslash command!

- Command to return schema size
- We'll use backslash y, because y-not
- Basic structure of describe.c function:

```
bool listSchemaSize(const char *pattern)
{
    initPQExpBuffer(&buf);
    printfPQExpBuffer(&buf, "the query");
    appendPQExpBufferStr(&buf, "more query");
    if (pattern)
        processSQLNamePattern(...)
    appendPQExpBufferStr(&buf, "group by");
    appendPQExpBufferStr(&buf, "order by");
    PSQLexec(buf.data);
    termPQExpBuffer(&buf);
    printQuery(...)
    PQclear(res);
    return true;
}
```

Query for backslash command!

- General query structure:

```
SELECT nspname,
       pg_size_pretty(
     sum(
   pg_total_relation_size(
 quote_ident(nspname) || '.' || quote_ident(relname)
   )))
FROM pg_namespace JOIN pg_class
     ON (pg_namespace.oid = pg_class.relnamespace)
WHERE relkind = 'r'
GROUP BY nspname
```

Variables needed for listSchemaSize

```
/*
 * listSchemaSize
 *
 * for \y
 */
bool
listSchemaSize(const char *pattern)
{
    PGresult    *res;
    PQExpBufferData buf;
    printQueryOpt myopt = pset.popt;
```

Build up Query

```
initPQExpBuffer(&buf);

printfPQExpBuffer(&buf,
                  "SELECT nspname as \"%s\",\n"
                "       pg_catalog.pg_size_pretty(pg_catalog.sum(\n"
                "         pg_catalog.pg_total_relation_size(\n"
                "           pg_catalog.quote_ident(nspname)\n"
                "           || '.' ||\n"
                "           pg_catalog.quote_ident(relname)))) as \"%s\"\n",
                  gettext_noop("Name"),
                  gettext_noop("Size"));
appendPQExpBufferStr(&buf,
             "\nFROM pg_catalog.pg_namespace JOIN pg_catalog.pg_class\n"
             "       ON (pg_namespace.oid = pg_class.relnamespace)\n"
                     "   WHERE relkind = 'r'");
```

Build up Query - pattern and group/order by

```
if (pattern)
    processSQLNamePattern(pset.db, &buf, pattern, true, false,
                          NULL, "pg_namespace.nspname", NULL, NULL);

appendPQExpBufferStr(&buf,
                     "\nGROUP BY nspname");

appendPQExpBufferStr(&buf, "\nORDER BY 1;");
```

Execute and print query results

```
  res = PSQLexec(buf.data);
  termPQExpBuffer(&buf);
  if (!res)
      return false;

  myopt.nullPrint = NULL;
  myopt.title = _("List of schema sizes");
  myopt.translate_header = true;

  printQuery(res, &myopt, pset.queryFout, pset.logfile);

  PQclear(res);
  return true;
}
```

Link it into command.c

- Command to return schema size
- We'll use backslash y, because y-not
- Basic structure of describe.c function:

```
/* \y is list schema size */
else if (strcmp(cmd, "y") == 0)
{
    char       *pattern;

    pattern = psql_scan_slash_option(scan_state,
                                     OT_NORMAL, NULL, true);

    success = listSchemaSize(pattern);

    if (pattern)
        free(pattern);
}
```

Next steps...

- Test it!
- Add into help.c, slashUsage()
- Update psql SGML documentation:
- doc/src/sgml/ref/psql-ref.sgml

libpq

libpq - client side

- Lives in src/interfaces/libpq
- "fe-" means "frontend"
- Implements the PostgreSQL protocol, client side

Major libpq/client components

- fe-auth.c - Send auth, get local username
- fe-connect.c - Handles connection setup to PG
- fe-exec.c - Send/receive query/data
- fe-misc.c - Low-level put/get routines
- fe-print.c - Pretty print query results
- fe-protocol3.c - Handles speaking moden PG protocol
- fe-secure.c - Handles encrypted/SSL communication
- fe-secure-openssl.c - OpenSSL wrapping for SSL
- libpq-events.c - libpq "events" API
- pqexpbuffer.c - String data type

Other libpq components

- fe-protocol2.c - Very old protocol
- fe-lobj.c - Large Object support (not bytea)
- pthread-win32.c - Partial pthreads implementation for Win32
- win32.c - Win32 helper routines

libpq - server side

- Lives in src/backend/libpq
- "be-" means "backend"
- Implements the PostgreSQL protocol, server side

libpq/server components

- auth.c - Handles auth with the client
- be-secure.c - Handles encrypted/SSL communication
- be-secure-openssl.c - OpenSSL wrapping for SSL
- crypt.c - Lookup PW in pg_authid, check it
- hba.c - Find HBA entry for connection, get auth method
- ip.c - IP address lookup/comparison routines
- md5.c - Low-level md5 routines
- pqcomm.c - Low-level communication routines
- pqformat.c - Get/send various types of data, text or binary
- pqmq.c - Support protocol conversation through shm_mq
- pqsignal.c - Handles blocking/unblocking signals
- be-fsstubs.c - Large Object handling

bin

- initdb - Initialize the database
- pg_archivecleanup - Cleans up old WAL, when not needed
- pg_basebackup - Take an online backup using replication
- pgbench - Performance benchmarking tool
- pg_config - Provides info about installed PG
- pg_controldata - Display control info about a cluster
- pg_ctl - Control a PG instance (start, stop, restart)
- pg_dump - Logically dump out data and structures from PG
- pgevent - Logging to Windows Event Log
- pg_resetxlog - Zero's out XLOG, can rebuild pg_control
- pg_rewind - "Remaster" old master to be new follower

bin

- pg_test_fsync - Test system fsync support
- pg_test_timing - Test overhead/monotonicity of timing calls
- pg_upgrade - In-place or binary copy major rev upgrade tool
- pg_xlogdump - Decode and display WAL/XLOG data
- scripts - simple wrapper commands:
  - clusterdb
  - createdb
  - createlang
  - createuser
  - dropdb
  - droplang
  - dropuser
  - pg_isready
  - reindexdb
  - vacuumdb

initdb

- Originally a shell script!
- Creates template1, then copies it to template0 and postgres
- Runs postgres in bootstrap mode, feeding data and commands
- Data comes from postgres.bki file
- Commands included in initdb.c and in .sql files
- BKI file generated from src/backend/catalog
- Invalid data in catalog .h files can cause initdb to fail
- SQL files are 'system_views.sql' and 'information_schema.sql'

pg_archivecleanup

- pg_archivecleanup.c - Routines to run the cleanup
- Includes src/backend/access/xlog_internal.h
- xlog_internal.h provides XLOG structures, #define's
- Relatively simple

pg_basebackup

- Actually three binaries included in pg_basebackup:
  - pg_basebackup - Take online backups
  - pg_receivexlog - Talks replication protocol to get XLOGs
  - pg_recvlogical - Receive logically decoded (via a slot) data
- pg_basebackup.c - Main routine, handles backup tar file
- pg_receivexlog.c - Talks replication protocol to get XLOGs
- pg_recvlogical.c - For logical decoding
- receivelog.c - Receive transaction log via streaming protocol
- streamutil.c - Utility functions used by all three utilities
- xlog_internal.h also used here

pgbench

- exprscan.l - Lexical scanner for pgbench expression language
- exprparse.y - Bison grammar for pgbench expression syntax
- pgbench.c - Main program
- Nice example of a utility with a simple language parser

pg_config

- pg_config.c - Main program
- Very simple
- Basically returns information from src/include/port.h

pg_controldata

- pg_controldata.c - Main program
- Reads $PGDATA/global/pg_control
- Uses lots of headers to minimize code duplication
- src/include/
  - access/xlog.h
  - access/xlog_internal.h
  - catalog/pg_control.h
  - postgres.h (not the usual postgres_fe.h)

pg_ctl

- pg_ctl.c - Main program
- Starts/stops/restart PG
- Includes routines to see if PG is alive
- Also handles promotion of follower to master
- Quite a bit of Windows-specific code also

pg_dump

- pg_dump.c - Main program for pg_dump
- pg_dumpall.c - Main program for pg_dumpall
- pg_restore.c - Main program for pg_restore
- common.c - Catalog lookup functions
- compress_io.c - Compression routines
- dumputils.c - Routines common to pg_dump and pg_dumpall
- parallel.c - Parallel support routines for pg_dump
- pg_dump_sort.c - Sort definitions for objects
- pg_backup_db.c - Connect/reconnect to DB
- pg_backup_utils.c - Routines common to pg_dump/restore
- pg_backup_archiver.c - Generic archive routines
- pg_backup_custom.c - Custom output format
- pg_backup_directory.c - Directory output format
- pg_backup_null.c - Used to generate plain SQL script

pg_dump

- Backup formats implemented via one interface
- Init function called to set up functions to use
- common.c pulls structure information about each object type then "dumps" it by creating ArchiveEntry's
- ArchiveEntry() creates entry for appropriate type of backup
- New objects require pg_dump support
- Mainly requires adding support to common.c
- New pg_dump formats should be pg_dump_format.c

pgeventlog

- Builds as a library
- Provides glue between backend and pg_ctl and Windows event log

pg_resetxlog

- pg_resetxlog.c - Main program
- Similar to pg_controldata - lots of backend headers used
- Also possible for it to rebuild pg_control itself
- Interesting headers included:
  - access/transam.h
  - access/tuptoaster.h
  - access/multixact.h
  - access/xlog.h
  - access/xlog_internal.h
  - catalog/catversion.h
  - catalog/pg_control.h
  - common/fe_memutils.h
  - common/restricted_token.h
  - storage/large_object.h

pg_rewind

- pg_rewind.c - Main program
- copy_fetch.c - Copy data using filesystem
- datapagemap.c - Keep track of changed data pages
- fetch.c - Generic fetch API, used by copy_fetch.c and libpq_fetch.c
- filemap.c - Keep track of changed files
- file_ops.c - Helper routines for writing to target dir
- libpq_fetch.c - Copy data using libpq
- logginc.c - Logging routines
- parsexlog.c - Read XLOG data, uses XLOG headers, etc
- timeline.c - Read timeline's history file

pg_test_*

- pg_test_fsync.c - Simply tests different fsync methods
- pg_test_timing.c - Tests how much overhead gettimeofday() costs and that it is always increasing

pg_upgrade - Main components

- pg_upgrade.c - Main program
- check.c - Checks run against old cluster to ensure clean upgrade
- controldata.c - Compares old and new control data
- dump.c - Generate dump of old cluster using pg_dumpll
- function.c - Checks C-language extensions and libraries
- info.c - Get info to map old files to new files
- page.c - Per-page conversion routines
- parallel.c - Routines to run parallel operations
- relfilenode.c - Handles copy/link of relation files
- tablespace.c - Get tablespace info, init new tablespaces
- version.c - Routines specific to certain PG versions

pg_upgrade - Supporting components

- server.c - General PG server connect, start/stop, routines
- option.c - Option handling
- exec.c - Routines for executing other programs, like pg_dumpall
- file.c - Low-level routines for copying and hard-linking files
- util.c - Utility routines, logging functions

backend Overview

Components of the backend (src/backend/...)

```
access      - Methods for accessing different types of data
              (heap, btree indexes, gist/gin, etc).
bootstrap   - Parse Back-End Interface files (for catalog)
catalog     - Routines used for modifying objects in pg_catalog
commands    - User-level SQL commands (CREATE/ALTER TABLE, etc)
executor    - Runs queries after planning / optimization
foreign     - Handles Foreign Data Wrappers, user mappings, etc
lib         - "General Purpose" / "Misc" functions
libpq       - Backend interface to talk to libpq
main        - Determines backend process startup / subsystems
nodes       - Node handling, build, copy, compare
optimizer   - Implements the costing system and generates plans
```

backend Overview continued

Components of the backend (src/backend/...)

```
parser      - Lexer and Grammar, how PG understands the queries
port        - Backend-specific platform-specific hacks
postmaster  - "main" PG process that always runs,
              answers requests, hands off connections
regex       - Henry Spencer's regex library, also used by TCL,
              maintained more-or-less by PG now
replication - Code for handling replication, WAL shipping
rewrite     - Query rewrite engine, used with RULEs, views
snowball    - Snowball stemming, used with full-text search
storage     - Storage layer, handles most direct file i/o and LO
tcop        - "Traffic Cop"- gets the actual queries, runs them
tsearch     - Full-Text Search engine
utils       - Cacheing system, memory manager, ACLs
```

backend components

PG-specific ways to do

- Memory management
- Error logging / cleanup
- Linked lists
- Catalog lookups
- Nodes / Various trees
- Datums
- Code Style
- Patch submission process

Memory management

- Nearly all memory allocated is tracked as part of a context
- Allocations happen through palloc()
- Contexts exist for different operations and lifetimes
  - CurrentMemoryContext - what palloc() will use
  - TopMemoryContext - Backend Lifetime (forever)
  - Per-Query Context
  - Per-Tuple Context
  - Function-call Contexts

Logging from PG

- Use ereport() with errcode() and errmsg()
- error level and errmsg() are required
- PG has a style guide for error messages
- ERROR or higher and PG will handle most cleanup
- Transaction rollback handled by ereport()
- Memory deallocation handled by ereport()

```
if (gzwrite(cstate->copy_gzfile, fe_msgbuf->data,
            fe_msgbuf->len) != fe_msgbuf->len)
    ereport(ERROR,
            (errcode_for_file_access(),
             errmsg("could not write to COPY file: %m")))
```

Catalog Lookups

- SysCache lookups with 'SearchSysCache'
- Defined in utils/cache/syscache.c
- Also some convenience routines in lsyscache.c

Nodes

- Various trees exist based on Nodes
- Each node has a 'type' plus appropriate data
- 'type' is stored in the node, allows IsA() testing
- Backend memory only, never out on disk, etc
- Create nodes using makeNode(TYPE)
- Node types defined in include/nodes/nodes.h
- make / copy / equality funcs in backend/nodes/

Tuples

- Heap Tuple defined in include/access/htup.h
- HeapTupleData is in-memory construct
- Provides length of tuple, pointer to header
- Pointer to disk buffer (must be pin'd)
- Could be empty
- May be a single palloc'd chunk
- Could be independently allocated
- Minimal Tuple structure (for hashing, etc)

Tuples (more)

- HeapTupleHeaderData and friends in htup_details.h
- Number of attributes
- Provides various flags (NULL bitmap, etc)
- Data follows the header (not in the struct)
- Lots of macros for working with tuples in detail

TOAST

- Large values can be compressed
- May also get "TOASTed" and moved to "toast" table
- Handled as a stored-out-of-line Datum
- Need to be careful with variable length Datums
- Typically try to avoid de-TOASTing Datums until necessary

Other subsystems

- Many things have already been done
- Eg: linked list implementation (llist.h)
- Generalized code should go in common area
- Look at existing code
- Real examples help immensely
- Portability considerations

Regression testing

- src/test/regress
- sql contains simple scripts to run
- expected contains expected results from scripts
- input are templates to generate sql files
- output are templates for generated scripts
- schedules are which tests to run
- parallel defines sets of tests to run in parallel
- serial are run serially
- serial run by pg_upgrade

contrib module structure

- PostgreSQL-included backend extensions
- Each has similar structure
- Regression tests supported for contrib also
- General structure of contrib modules:
  - Makefile to build contrib module
  - .c/.h for contrib module
  - sql directory for regression scripts
  - expected directory for regression script results
  - .control file with module information
  - –1.0.sql script to create functions, etc
  - Additional .h/.c files as necessary

Writing a contrib module

- Copy existing one!
- Very simple one exists- passwordcheck.c

```
cd /home/training/pg/src/feature/contrib
cp -a passwordcheck mymodule
vi Makefile
cd mymodule
mv passwordcheck.c mymodule.c
vi Makefile
vi mymodule.c
```

Using hooks

- Many, many hooks exist in PostgreSQL
- Allows module to gain control at certain point
- passwordcheck uses "check_password_hook"
- Module's _PG_init() called on module load
- hooks can be chained, or not
- Anything loaded is dangerous- just like backend C code

Loading modules

- hook-only can be loaded via shared_preload_libraries
- Complex modules are created with CREATE EXTENSION
- CREATE EXTENSION requires .control, .sql script
- Objects created during .sql are tracked as part of extension
- Upgrade .sql scripts can be provided
- eg: dblink–1.0–1.1.sql

Query handling

- Queries pass through many pieces to be executed
- psql receives query directly from user
- libpq used by psql to send query to server
- server receives query via libpq (backend)
- server parses query, plans query, executes query
- Results sent to client via libpq (backend)
- Results received by client via libpq
- Results displayed by psql

psql

- Receives query via input.c
- Sends query to libpq via common.c / SendQuery
- Receives query results via common.c / ProcessResults
- Prints results with print.c

libpq

- Receives query via fe-exec.c / PQexec
- Sends query to backend via fe-exec.c / PQsendQuery
- Receives query results via fe-protocol.c / pqParseInput3()
- fe-exec.c gets results via parseInput
- Results returned to caller via PQresults

Simple query

- Query message sent to backend
- backend responds with RowDescription
- Followed by DataRow messages, for all rows
- Next is CommandComplete
- Finally ReadyForQuery
- Multiple RowDescription/DataRow/CommandComplete possible
- One for each SQL query in string sent by client

Extended query

- Parse message sent first, includes placeholders
- Backend responds with ParseComplete
- Bind message provides values for placeholders
- Execute message kicks off query
- backend responds with RowDescription
- Followed by DataRow messages, for all rows
- Next is CommandComplete
- Frontend should issue Sync message at end of Extended messages
- Finally ReadyForQuery

backend - traffic cop

- src/backend/tcop/postgres.c
- PostgresMain() reads command from protocol layer
- exec_simple_query() called to execute query
- Query parsed using pg_parse_query()
- Using result of parsing, analyze and rewrite query
- Plan query using pg_plan_queries
- Calls planner(), plans/optimizes query
- Then calls ExecutorRun via Portal
- Receiver created and used for results
- End command
- Loop back up for next query

backend - parser

- src/backend/parser
- raw_parser() called from pg_parse_query()
- Runs bison/flex generated parser

backend - planner

- Actually planner and optimizer
- Lives in src/backend/optimizer
- Entry is plan/planner.c / planner()
- Heavy lifting by subquery_planner()
- Followed by grouping_planner()

backend - executor

- Handles executing the query and returning results
- Lives in src/backend/executor
- Entry is execMain.c / ExecutorRun()
- Calls down to ExecutePlan()
- Then ExecProcNode() - execute node, return a tuple
- Continues for specified number of tuples, or all

backend - tuplestore

- Receive of tuples can be a tuplestore
- Exists in memory, while memory is available
- Spills over to disk when out of memory

backend - storage

- src/backend/storage
- Only one storage manager today- smgr
- General concept kept for now
- smgr.c provides interface for users
- md.c maps smgr interface to kernel calls
- file/fd.c manages set of open file descriptors
- Do not want to hit open file limit

Create a Branch

- Script provided to create new branch
- make_branch.sh - pass new branch name, and branch to go from
- Uses git-new-workdir for new branch
- git-new-workdir creates new directory which is linked to main git repo
- Minimized additional disk space requires

Commiting in Git

- Add files to commit using 'git add'
- commit files added using 'git commit'
- commit all changed files with 'git commit -a'
- Requires a commit message
- Short commit message can be passed with -m
- 'git commit -am "message"'

Fix-up Commits

- git commit -a –fixup HEAD
- Handy command- aliases as 'gcf'
- Fixup commits do not require a message
- Will be automatically set for squash

Squashing Commits

- Combine commits together
- Prefer to squash most commits into single, large, commit
- Also generally simpler/easier to review larger patches
- 'git rebase -i –autosquash'
- aliased as grbi
- Opens editor to choose actions
- Generally, 'reword' first, 'squash' rest
- Opens editor for rewording commit message

Git format-patch

- Generates patch from commits
- Patch can be emailed, etc
- 'git format-patch @u –stdout'
- aliased as gfp

Git diff

- Provides diff of changes from last commit
- 'git diff'
- aliased as 'gd'
- Diff against upstream instead
- 'git diff @u'
- aliased as gdu
- Checks also available
- 'git diff –check' - alias is gdc
- 'git diff –check @u' - alias is gdcu

So you have an idea...

Where to begin?

- Depends on your idea, but I prefer the parser
- Grammar informs the design
- Also one of the hardest items to get agreement on

Grammar is in src/backend/parser/

- scan.l - lexer, handles tokenization
- gram.y - actual grammar
- Built with flex (lexer) and bison (parser)
- Rarely have to change the lexer (be careful!)

Modifying the Grammar

Grammar is a set of productions

- "main" is the 'stmt' production
- Lists all the top-level commands
- Each is then its own production

```
stmt :
            AlterEventTrigStmt
            | AlterDatabaseStmt
            | AlterDatabaseSetStmt
            ...
            | CopyStmt

CopyStmt :
  COPY opt_binary qualified_name opt_column_list opt_oids
       copy_from opt_program copy_file_name copy_delimiter
```

Modifying CopyStmt

- Add it into the COPY production
- Modify the C template code as needed
  - C code is extracted by bison
  - Run through a set of changes (eg: changes "$3")
  - Compiled as part of the overall parser (gram.c)
- Remember to update the keywords list (kwlist.h)
- Also remember to add to unreserved_keywords
- Try to avoid creating new *reserved* keywords

Adding an option to COPY

```
--- a/src/backend/parser/gram.y
+++ b/src/backend/parser/gram.y
@@ -521,8 +521,8 @@ static void processCASbits(int cas_bits, int location, const char * constrType,
-   COMMITTED CONCURRENTLY CONFIGURATION CONNECTION CONSTRAINT CONSTRAINTS
-   CONTENT_P CONTINUE_P CONVERSION_P COPY COST CREATE
+   COMMITTED COMPRESSED CONCURRENTLY CONFIGURATION CONNECTION CONSTRAINT
+   CONSTRAINTS CONTENT_P CONTINUE_P CONVERSION_P COPY COST CREATE
@@ -2403,6 +2403,10 @@ copy_opt_item:
                 {
                     $$ = makeDefElem("header", (Node * )makeInteger(TRUE));
                 }
+            | COMPRESSED
+                {
+                    $$ = makeDefElem("compressed", (Node * )makeInteger(TRUE));
+                }
             | QUOTE opt_as Sconst
                 {
                     $$ = makeDefElem("quote", (Node * )makeString($3));
@@ -12471,6 +12475,7 @@ unreserved_keyword:
             | COMMITTED
+            | COMPRESSED
             | CONFIGURATION
```

What about the code?

- COPY has a function to process options
- Surprise, it's called "ProcessCopyOptions"
- COPY is defined in backend/commands/copy.c
- COPY state info
- Local state structure CopyStateData also in copy.c
- Not in a .h because only COPY needs it
- Define structures in .c files near the top

Option handling in COPY

```
@@ -109,6 +119,7 @@ typedef struct CopyStateData
    bool        binary;         /* binary format? * /
+   bool        compressed;     /* compressed file? * /
    bool        oids;           /* include OIDs? * /
@@ -889,6 +1186,20 @@ ProcessCopyOptions(CopyState cstate,
        }
+       else if (strcmp(defel->defname, "compressed") == 0)
+       {
+#ifdef HAVE_LIBZ
+           if (cstate->compressed)
+               ereport(ERROR,
+                       (errcode(ERRCODE_SYNTAX_ERROR),
+                        errmsg("conflicting or redundant options")));
+           cstate->compressed = defGetBoolean(defel);
+#else
+           ereport(ERROR,
+                   (errcode(ERRCODE_SYNTAX_ERROR),
+                    errmsg("Not compiled with zlib support.")));
+#endif
+       }
        else if (strcmp(defel->defname, "oids") == 0)
```

Other changes

- Many more changes to copy.c needed
- New 'COMPRESSED' state
- Tracking gzFile instead of FILE*
- Using gzread / gzwrite instead of read/write
- Data input/output handling
- All data handled with 2 buffers, uncompressed and compressed

Diffstat

```
doc/src/sgml/ref/copy.sgml          |   12 ++
src/backend/commands/copy.c         |  458 ++++++++++++++++++++++++++++++++++++-----
src/backend/parser/gram.y           |    9 +-
src/backend/storage/file/fd.c       |   97 ++++++++++
src/include/parser/kwlist.h         |    1 +
src/include/storage/fd.h            |    9 ++
src/test/regress/input/copy.source  |   20 +++
src/test/regress/output/copy.source |   18 +++
8 files changed, 583 insertions(+), 41 deletions(-)
```

- doc/src - Documentation updates
- Modify fd.c for compressed files
- fd.c provides file descriptor cacheing
- Added: AllocateFileGz, FreeFileGz
- src/test/regress - New regression tests

COPY PIPE

- Follow the mailing lists
- Watch for others working on similar capabilities
- Try to think about general answers, not specific
- Be supportive of other ideas and approaches
- Send and receive COPY data from program instead
- E.g. for gzipped files
- postgres=# COPY t FROM PROGRAM 'zcat /tmp/t.csv.gz'

Choose grammar to use

- Strongly suggest, initially, simple new command
- Implement grammar first
- Add nodes and structures required for grammar
- Implement actual command second
- Follow existing style for where code goes

ALTER TABLE .. FORCE ROW SECURITY

- Another feature patch to review
- Relatively simple
- Includes grammar changes
- Also modified catalog tables
- Update catalog version- requires initdb

Mailing Lists

Submitting Patches

- Use context diff or git-diff
- Read the actual patch before posting
- Email -hackers the patch
- Include description of the patch
- Don't forget regression tests, pg_dump support, documentation
- Register patch on commitfest.postgresql.org

Commitfest Application

- New patches submitted via commitfest.postgresql.org
- Patch should first be emailed to -hackers mailing list
- One on -hackers, register patch in commitfest

Patch Status

- Attempt to track what the current status of the patch is
- "Needs Review" - Waiting for someone to review the patch
- "Waiting on Author" - For various reasons
- "Ready for Committer" - Next level review
- "Returned with Feedback" - Essentially bumped to next CF
- "Rejected" - Generally means not right approach, etc

Patch Review

- Important that patches are reviewed before being applied
- Helpful to have non-committers do initial review
- When submitting a patch, consider what patch to review
- Hopefully, other authors will review your patch

Patch Commit

- Once patch is in "Ready for Committer"...
- Hopefully it gets committed!
- May be applied, returned, rejected by committer
- Commit message will include attribution

Thank You!

Thank You!

stephen@crunchydata.com